

Unclassified  
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

100-1000

1a. REPORT SECURITY CLASSIFICATION Unclassified		1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY SELECT		3. DISTRIBUTION/AVAILABILITY OF REPORT Unlimited	
2b. DECLASSIFICATION/DOWNGRADING DATE 29 1990		5. MONITORING ORGANIZATION REPORT NUMBER(S)	
1. PERFORMING ORGANIZATION REPORT NUMBER(S) TR 90-1164		7a. NAME OF MONITORING ORGANIZATION Office of Naval Research	
a. NAME OF PERFORMING ORGANIZATION Cornell University		6b. OFFICE SYMBOL (if applicable)	
c. ADDRESS (City, State, and ZIP Code) Department of Computer Science Upson Hall, Cornell University Ithaca, NY 14853		7b. ADDRESS (City, State, and ZIP Code) 800 North Quincy St. Arlington, VA 22217-5000	
1. NAME OF FUNDING/SPONSORING ORGANIZATION Office of Naval Research		8b. OFFICE SYMBOL (if applicable)	
11. TITLE (Include Security Classification) Towards Derivation of Real-Time Process-Control Programs		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N000014-86-K-0092	
12. PERSONAL AUTHOR(S) Keith Marzullo and Fred B. Schneider		10. SOURCE OF FUNDING NUMBERS	
13a. TYPE OF REPORT Interim		13b. TIME COVERED FROM _____ TO _____	
14. DATE OF REPORT (Year, Month, Day) October 11, 1990		15. PAGE COUNT 12	
16. SUPPLEMENTARY NOTATION			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) real-time software, process-control software, weakest preconditions, auxiliary variables, program verification, (etc.)	
19. ABSTRACT (Continue on reverse if necessary and identify by block number) The use of weakest-precondition predicate transformers in the derivation of sequential, process-control software is discussed. Only the extension to Dijkstra's calculus for deriving ordinary sequential programs was found to be necessary: function-valued auxiliary variables. These auxiliary variables are needed for reasoning about states of a physical process that exist during program transitions. <i>Keyed: Conf</i>			
<b>DISTRIBUTION STATEMENT A</b> Approved for public release; Distribution Unlimited			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION	
22a. NAME OF RESPONSIBLE INDIVIDUAL Fred B. Schneider		22b. TELEPHONE (Include Area Code) (607) 255-9221	
22c. OFFICE SYMBOL			

Towards Derivation of  
Real-Time Process-Control Programs

Keith Marzullo \*  
Fried B. Schneider \*\*

Department of Computer Science  
Cornell University  
Ithaca, New York 14853

October 11, 1990

ABSTRACT

The use of weakest-precondition predicate transformers in the derivation of sequential, process-control software is discussed. Only one extension to Dijkstra's calculus for deriving ordinary sequential programs was found to be necessary: function-valued auxiliary variables. These auxiliary variables are needed for reasoning about states of a physical process that exist during program transitions.

---

\* Supported in part by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593 Contract N00140-87-C-8904 and a grant from Xerox Corporation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.

\*\* Supported in part by the Office of Naval Research under contract N00014-86-K-0092, the National Science Foundation under Grants No. CCR-8701103 and CCR-9003440, and Digital Equipment Corporation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.

## 1. Introduction

For the past few years, we have been exploring the use of assertional reasoning in the construction of process-control software. Our intent was to employ an existing method, perhaps with a few extensions, and systematically derive process-control programs from specifications. Use of an existing method had both a scientific and a pragmatic motivation. The scientific motivation was based on our expectation that the difficulties we encountered by using an existing method would provide insights into what distinguishes process-control programs from ordinary sequential and concurrent programs. The pragmatic motivation for using an existing method was that extending a well understood method was likely to be easier than developing a new method.

Our investigations have been structured as a series of experiments. Each experiment is based on a simple process-control problem that (we feel) epitomizes some aspect of process-control programming. We started with the simplest process-control problem imaginable—a sequential control-program running on a single, fault-free processor. By reading sensors and writing to actuators, this program controls an on-going physical process. Solving this problem required reasoning about control-program execution times, something that has long been considered an integral part of process-control programming. We were well aware, however, that any conclusions from this experiment would have to be regarded as tentative. By considering a sequential control-program, problems arising due to resource contention were being ignored; and by assuming a fault-free processor, complications associated with implementing fault-tolerance were being ignored.

Simplifying assumptions notwithstanding, our first experiment did lead to some insights about the use of assertional reasoning in writing process-control programs. These insights are the subject of this paper. In section 2, we describe extensions to Dijkstra's weakest-precondition calculus [D76] [G84] that we found necessary for deriving sequential process-control programs. Section 3 illustrates the use of these extensions and the calculus by giving an example derivation of a control program. Conclusions appear in section 4.

## 2. Using Weakest Preconditions with Physical Processes

Process-control problems are often specified in terms of restrictions on permissible states of some given physical system. To prevent these restrictions from being violated, a control program is constructed. By setting actuators to manipulate the process being controlled, the control program ensures that none of the proscribed states is ever entered. The actions of a control program are, therefore, closely linked to the state of the physical process being controlled. Consequently, when deriving a control program, it is necessary that we be able to reason about both the program state and the state of the physical process being controlled.

Assertional methods for deriving programs are based on manipulating logical formulae, called *assertions*, that characterize sets of program states. One way to employ assertional methods in the design of a process-control program is to augment the program state space so that it includes

information about the state of the physical process being controlled. Doing so, however, requires extending the rules used to reason about program execution:

- (1) During execution of a program statement, changes occur to the state of the physical process being controlled. Rules characterizing the effects of program execution must, therefore, be modified to reflect these other state changes.
- (2) Statements whose execution involves interaction with sensors and/or actuators must be axiomatized as rules relating states before, during, and after execution.

The remainder of this section discusses these extensions.

### Reality Variables

The state space of physical system is usually defined by a collection of state components, each of which is indexed by some independent parameters. For example, the state of a railroad train at a time  $T$  can be characterized by its position  $X(T)$ , its speed  $V(T)$ , and its acceleration  $A(T)$ . Note that the choice of time as the independent parameter is arbitrary. If its velocity is always greater than 0, then a train at position  $X$  could equally well be described by time  $T(X)$ , speed  $V(X)$ , and acceleration  $A(X)$ . As physicists learned long ago, quantities that are convenient for the task at hand should be selected as the independent parameters.

The state space of a program can be augmented to include the state of a physical process by postulating additional program variables. For each state component  $Q_i$ , we add to the program state space a function-valued program variable  $q_i$ , called a *reality variable*.<sup>1</sup> Each reality variable replicates (in the program's state space) information about a physical system during program execution. Initially, the domain of a reality variable  $q_i$  will be empty; as the independent parameter  $P_i$  for  $Q_i$  changes, the domain of  $q_i$  is extended to include the values over which  $P_i$  has ranged. Reality variables are entirely fictional. They allow us to describe and reason about the state of a physical system by using assertions, but they are not actually maintained in memory. Thus, they are a form of auxiliary variable [C73].

In defining and manipulating expressions involving function-valued program variables, like reality variables, it will be convenient to have some notation. Following [G84], given a function  $f$  with domain  $dom(f)$ , the function expression

$$(f; x \in D : g(x))$$

is defined to be a function whose domain is  $dom(f) \cup D$  and whose value at any point  $a$  in this domain is:  $g(a)$  if  $a \in D$  and  $f(a)$  otherwise. As a notational convenience, we define:

---

<sup>1</sup>In the sequel, we use upper-case identifiers to denote physical state components and the corresponding lower case identifier to denote the reality variables that model them.

$$(f; x \in D : g(x); x \in D' : h(x)) = ((f; x \in D : g(x)); x \in D' : h(x)).$$

And, in specifying a domain, we use the notation *low* .. *high* to denote the set  $\{a \mid low \leq a \leq high\}$ .

### Preserving the Fiction: Updating Reality Variables

The state of a physical system is changed by a physical process. Typically, these changes are characterized by a set of equations relating the values of various state components to current and recent values. We cannot expect a physical process to update the reality variables being used by the control program for modeling the state of a physical system. And, since the weakest-precondition calculus is based on the presumption that all changes to the truth of an assertion (being evaluated on the program state) are the result of program execution, we have no choice but to regard the program itself as performing updates to reality variables. By using the same laws of Physics that characterize changes to the state of the physical process, it is possible for program statements to compute updates to the reality variables.

Consider some physical state component  $Q(P)$  being modeled by a reality variable  $q(p)$ , and suppose that as long as no actuator changes during some interval from  $P$  to  $P + \delta$ , changes to  $Q$  are characterized by the following equation.

$$(2.1) \quad Q(P + \delta) = \mathcal{F}(Q(P), \delta) \text{ for } 0 \leq \delta \leq \delta$$

Let  $\langle S \rangle_\delta$  denote a statement whose execution coincides with a change of  $\delta$  of parameter  $P$ . Then, execution of  $\langle S \rangle_\delta$  is equivalent to executing  $S$  and, as part of the same atomic action, changing  $p$  and  $q$  in accordance with (2.1). This state change can be modeled by a program fragment:

$$\langle S \rangle_\delta: S; p, q := p + \delta, (q; i \in p .. p + \delta: \mathcal{F}(q(p), i - p))$$

Using the weakest-precondition predicate transformers for multiple-assignment and statement composition, we obtain the following predicate transformer characterization for  $\langle S \rangle_\delta$ .

$$\begin{aligned} & \text{wp}(\langle S \rangle_\delta, R) \\ &= \text{wp definition of ";" *} \\ &= \text{wp}(S, \text{wp}(p, q := p + \delta, (q; i \in p .. p + \delta: \mathcal{F}(q(p), i - p)), R)) \\ &= \text{wp definition of ":" *} \\ &= \text{wp}(S, R_{p+\delta}^p, (q; i \in p .. p + \delta: \mathcal{F}(q(p), i - p))) \end{aligned}$$

Notice that when the independent parameter  $\delta$  in  $\langle S \rangle_\delta$  models the passage of time,  $\langle S \rangle_\delta$  is a statement that executes for  $\delta$  seconds. The definition of  $\text{wp}(\langle S \rangle_\delta, R)$  then asserts that after executing  $\langle S \rangle_\delta$  the current time has been incremented by  $\delta$  and all other reality variables have been updated as if  $\delta$  seconds had passed. However, our characterization of  $\langle S \rangle_\delta$  also allows the independent parameter  $\delta$  to be a quantity other than time, making it possible to reason in the coordinate system best suited for the problem at hand. Also notice that, according to our weakest precondition characterization of  $\langle S \rangle_\delta$ , an ordinary statement  $S$  must be regarded as being equivalent to  $\langle S \rangle_0$ . This is because

$$wp(\langle S \rangle_0, R) = wp(S, R)$$

holds.

To illustrate the use of  $wp(\langle S \rangle_\delta, R)$  in an actual process-control programming problem, suppose we are interested in controlling the speed of a railroad train. Define reality variable  $v(x)$  to be the speed of the train when at a given position  $x$ . From Newton's Laws of Motion, we know that if a train does not accelerate during an interval of  $\delta$  seconds, then reality variable  $v$  can be characterized by the following equation:

$$(2.2) \quad v(x+\Delta) = v(x) \text{ for } 0 \leq \Delta \leq v(x) * \delta$$

Thus, according to our definition for  $wp(\langle S \rangle_\delta, R)$ , we have the following weakest precondition characterization for a statement  $\langle S \rangle_\delta$  that takes duration  $\delta$  seconds and is executed while a train is not accelerating.

$$\begin{aligned} & wp(\langle S \rangle_\delta, R) \\ = & \quad \ll (2.2) \text{ and } wp \text{ definition for } \langle S \rangle_\delta \gg \\ & wp(S, R_{x+v(x)=\delta}^{x+v(x)=\delta}, (v; l \in x \dots x+v(x)=\delta; v(x))) \end{aligned}$$

### Interacting with a Physical Process

To have broad applicability, a method for reasoning about process-control programs must not restrict the types of sensors and actuators that it can handle. In methods (like the weakest-precondition calculus) where all state changes are attributed to execution of statements, rules for reasoning about the effects of sensors and actuators are derived by

- (1) modeling these effects as updates to reality variables, and then
- (2) using the rules provided for reasoning about ordinary statements to derive rules for the statements being modeled.

As long as the updates to reality variables correctly capture the effects of the sensor or actuator, the resulting rules will be sound and can be used to reason about how a control program interacts with the process it controls.

To illustrate how sensors and actuators are modeled, we return to railroad control. We consider an actuator  $go(t)$  and a sensor  $await(c)$ . Executing  $go(t)$  causes the train to start accelerating/decelerating with some maximum constant acceleration  $ACC$  (say) until target speed  $t$  is reached; execution terminates only when the train reaches its target speed.  $await(c)$ , if invoked while the train is not accelerating, delays execution of a program until the train is at location  $c$ .<sup>2</sup>

---

<sup>2</sup>If  $go(t)$  is the only actuator that can cause acceleration, then the condition that  $await(c)$  is never executed while the train is accelerating is equivalent to stipulating that a train is controlled by a single sequential program.

Define  $Vlen(u, t)$  to be the distance that a train travels while it is accelerating from speed  $u$  to target speed  $t$ :

$$Vlen(u, t) = \frac{1}{2}((u^2 - t^2)/(2*ACC))$$

And, define  $Vat(u, t, x)$  to be the speed of a train after having traveled  $x$  meters,  $0 \leq x \leq Vlen(u, t)$ , from the point at which it started accelerating from speed  $u$  to  $t$ :

$$Vat(u, t, x) = \begin{cases} \sqrt{u^2 + 2*x*ACC} & \text{if } u < t \\ \sqrt{u^2 - 2*x*ACC} & \text{if } u > t \\ u & \text{if } t = u \end{cases}$$

The effect of executing  $go(t)$  can be modeled as an update to reality variables  $x$  and  $v$ . The value of  $x$  is increased by  $Vlen(v(x), t)$  and the domain of  $v$  is extended to include  $x .. x + Vlen(v(x), t)$ :

$$go(t): x, v := x + Vlen(v(x), t), (v; l \in x .. x + Vlen(v(x), t); Vat(v(x), t, l - x))$$

Even though executing  $go(t)$  actually prescribes changes to actuators, this multiple-assignment statement model does provide a basis for calculating  $wp(go(t), R)$ :

$$\begin{aligned} & wp(go(t), R) \\ &= \text{«model of } go(t)» \\ &= wp(x, v := x + Vlen(v(x), t), (v; l \in x .. x + Vlen(v(x), t); Vat(v(x), t, l - x)), R) \\ &= \text{«wp definition of »} := \\ & R_{x+Vlen(v(x), t)}^{x, v} (v; l \in x .. x + Vlen(v(x), t); Vat(v(x), t, l - x)) \end{aligned}$$

Similarly,  $await(c)$  can be modeled by an alternative command:

$$await(c): \text{if } x \leq c \wedge 0 < v(x) \rightarrow x, v := c, (v; l \in x .. c; v(x)) \text{ fi}$$

Our model for  $await(c)$  updates reality variables  $x$  and  $v$  if  $x \leq c$  and  $0 < v(x)$  hold; otherwise, it delays forever. Using the weakest precondition for  $\text{if}$ , we can calculate a weakest precondition predicate transformer for  $await(c)$ :

$$\begin{aligned} & wp(await(c), R) \\ &= \text{«model of } await(c)» \\ &= wp(\text{if } x \leq c \wedge 0 < v(x) \rightarrow x, v := c, (v; l \in x .. c; v(x)) \text{ fi}, R) \\ &= \text{«wp definition of if»} \\ &= x \leq c \wedge 0 < v(x) \wedge (x \leq c \wedge 0 < v(x) \Rightarrow wp(x, v := c, (v; l \in x .. c; v(x)), R)) \\ &= \text{«wp definition of »} := \text{ and predicate logic»} \\ &= x \leq c \wedge 0 < v(x) \wedge R_{c, (v; l \in x .. c; v(x))}^{x, v} \end{aligned}$$

### 3. An Example

Other than the extensions mentioned above, the methodology of [D76] and [G84] for deriving ordinary sequential programs can be used, unchanged, for deriving sequential process-control programs. In this section, we illustrate that methodology with a simple railroad-control problem.

Railroad tracks are typically partitioned into segments, called *blocks*. Each block  $i$ , has an associated starting location  $b_i$  and ending location  $b_{i+1}$  where  $b_i \leq b_{i+1}$ , and a range of permissible speeds  $mn_i \dots mx_i$ , where  $0 \leq mn_i < mx_i$ . Desired is a program to control the speed of a point train<sup>3</sup> so that it travels from  $b_0$  to  $b_n$ , maintaining safe speeds along the way. Use `go(t)` and `await(c)`, as defined above, for interactions between a single sequential control program and the train.

The train has made a safe passage from location  $a$  to  $b$  provided the following holds.

$$Safe(a, b): a \dots b \subseteq \text{dom}(v) \wedge (\forall l: a \leq l \leq b: b_i \leq l \leq b_{i+1} \Rightarrow mn_i \leq v(l) \leq mx_i)$$

The first conjunct of  $Safe(a, b)$  asserts that the train has actually traveled from  $a$  to  $b$ , and the second conjunct asserts that the train's speed satisfied the restrictions associated with each block it occupied. Using  $Safe(a, b)$ , we can specify the above railroad control problem in terms of weakest preconditions:

$$(3.1) \quad x = b_0 \wedge v = (; b_0: v_0) \Rightarrow wp(S, Safe(b_0, b_n) \wedge x = b_n)$$

This formula constrains  $S$  to be a program that terminates with the train at location  $b_n$  after having traveled at safe speeds to get there, provided  $S$  is started with the train at location  $b_0$  traveling with speed  $v_0$ .<sup>4</sup>

Having formalized the specification for a correct control program  $S$ , we now proceed with the derivation. The universal quantifier in conjunct  $Safe(b_0, b_n)$  of the result assertion is a tip-off that a loop should be tried for  $S$ . Thus, we employ a standard heuristic from [G84]—replacing a constant by a variable—and derive a loop invariant from the result assertion. Replacing  $n$  in the result assertion by a new program variable  $h$  (for “here”) we get:

$$I: Safe(b_0, b_h) \wedge x = b_h \wedge 0 \leq h \leq n$$

Since  $I \wedge h = n$  implies result assertion  $Safe(b_0, b_n) \wedge x = b_n$ , we conclude that the loop guard must be  $h \neq n$  (or something that implies  $h \neq n$ ) and conjecture that  $S$  has the following structure:

---

<sup>3</sup>Assuming a point train is not a fundamental assumption, but will simplify some of what follows. By using a configuration space transformation [LP83], the control problem for a length  $L$  train can be transformed to a control problem for a point train on a track with additional blocks.

<sup>4</sup>If the conjunct  $x = b_n$  is omitted from the result assertion, then it would be permissible for control program  $S$  to terminate long after the train had passed point  $b_n$ . We have deemed such behavior to be unacceptable and have ruled it out.

$S: S_1 \{I\}$   
 do  $h \neq n \rightarrow \{I \wedge h \neq n\} S_2 \{I\}$  od  $\{h = n \wedge I\}$   
 $\{Safe(b_0, b_n)\}$

Program  $S$  will satisfy its specification provided we find statements  $S_1$  and  $S_2$  that satisfy the following specifications.

$$(3.2) \quad b_0 \wedge v = (; b_0; v_0) \Rightarrow wp(S_1, I)$$

$$(3.3) \quad I \wedge h \neq n \Rightarrow wp(S_2, I)$$

Formula (3.2) is the specification for the loop initialization; (3.3) is the specification for the loop body.

According to specification (3.2),  $S_1$  must establish  $I$ . Observe that an easy way to establish  $I$  is by setting  $h$  to 0. So, we use  $wp$  to calculate an assertion that must hold before executing  $h := 0$  in order for  $I$  to hold afterwards.

$$\begin{aligned}
 & wp(h := 0, I) \\
 &= \text{«wp definition of ":=»} \\
 & (Safe(b_0, b_n) \wedge x = b_n \wedge 0 \leq h \leq n) \\
 &= \text{«textual substitution»} \\
 & Safe(b_0, b_0) \wedge x = b_0 \\
 &= \text{«definition of } Safe(a, b) \text{»} \\
 & b_0 \in \text{dom}(v) \wedge mn_0 \leq v(b_0) \leq mx_0 \wedge x = b_0
 \end{aligned}$$

Notice that  $x = b_0 \wedge v = (; b_0; v_0)$ , the antecedent of specification (3.1) for  $S$ , implies  $wp(h := 0, I)$  only if  $mn_0 \leq v_0 \leq mx_0$ . Thus, executing  $h := 0$  establishes the loop invariant only under certain conditions—the initial speed of the train must be safe for travel in block  $b_0$ .

**Assumption 1.**  $mn_0 \leq v_0 < mx_0$

In retrospect, this requirement should not be surprising. What is worth noting, however, is that this implicit assumption was exposed simply by adhering to a rigorous calculus in deriving the program.

We have derived following structure for the loop.

$S: \{x = b_0 \wedge v = (; b_0; v_0) \wedge mn_0 \leq v_0 < mx_0\}$   
 $h := 0 \{I\}$   
 do  $h \neq n \rightarrow \{I \wedge h \neq n\} S_2 \{I\}$  od  $\{h = n \wedge I\}$   
 $\{Safe(b_0, b_n)\}$

We now refine  $S_2$ , the body of the loop. Based on our choice of guard, we know that the loop will terminate when  $h$  equals  $n$ . Initially,  $h$  is 0. Thus, for  $S_2$  to make progress towards termination,  $h$  must be increased; and for  $S_2$  to satisfy specification (3.3),  $S_2$  must reestablish  $I$ . To investigate the feasibility of increasing  $h$  by 1, we calculate  $wp(h := h + 1, I)$ .

$$\begin{aligned}
 & wp(h := h + 1, I) \\
 &= \text{«wp definition of ":=»}
 \end{aligned}$$

$$\begin{aligned}
& (Safe(b_0, b_h) \wedge x = b_h \wedge 0 \leq h \leq n)_{h+1}^n \\
= & \text{«textual substitution»} \\
& Safe(b_0, b_{h+1}) \wedge x = b_{h+1} \wedge 0 \leq h+1 \leq n \\
= & \text{«} a \leq b \leq c \Rightarrow (Safe(a, c) = (Safe(a, b) \wedge Safe(b, c))) \text{»} \\
& Safe(b_0, b_h) \wedge Safe(b_h, b_{h+1}) \wedge x = b_{h+1} \wedge 0 \leq h+1 \leq n
\end{aligned}$$

Since  $l \wedge h \neq n$  holds at the start of  $S_2$ , we know that the first and last conjuncts of  $wp(h := h+1, l)$  hold before  $S_2$  executes. We must, therefore, arrange for the remaining conjuncts to hold.

$$\begin{aligned}
& Safe(b_h, b_{h+1}) \wedge x = b_{h+1} \\
= & \text{«definition of } Safe(a, b) \text{»} \\
& b_h \dots b_{h+1} \subseteq \text{dom}(v) \wedge (\forall l: b_h \leq l \leq b_{h+1}: b_i \leq l \leq b_{i+1} \Rightarrow mn_i \leq v(l) \leq mx_i) \wedge x = b_{h+1} \\
= & \text{«} x = b_{h+1} \Rightarrow 0 \dots b_{h+1} \subseteq \text{dom}(v) \text{»} \\
& (\forall l: b_h \leq l \leq b_{h+1}: b_i \leq l \leq b_{i+1} \Rightarrow mn_i \leq v(l) \leq mx_i) \wedge x = b_{h+1} \\
= & \text{«predicate logic»} \\
(3.4) \quad & (\forall l: b_h \leq l \leq b_{h+1}: b_i \leq l < b_{i+1} \Rightarrow mn_i \leq v(l) \leq mx_i) \\
& \wedge \max(mn_h, mn_{h+1}) \leq v(b_{h+1}) \leq \min(mx_h, mx_{h+1}) \wedge x = b_{h+1}
\end{aligned}$$

We consider the final conjunct first. It is easy to establish by using  $\text{await}(b_{h+1})$ , so we compute:

$$\begin{aligned}
& wp(\text{await}(b_{h+1}); h := h+1, (3.4)) \\
= & \text{«wp calculus»} \\
& x \leq b_{h+1} \wedge 0 < v(x) \wedge ((\forall l: b_h \leq l \leq b_{h+1}: b_i \leq l < b_{i+1} \Rightarrow mn_i \leq v(l) \leq mx_i) \\
& \wedge \max(mn_h, mn_{h+1}) \leq v(b_{h+1}) \leq \min(mx_h, mx_{h+1}) \wedge x = b_{h+1})_{b_{h+1}}^v(v; l \in x \dots b_{h+1}; v(x)) \\
= & \text{«textual substitution»} \\
& x \leq b_{h+1} \wedge 0 < v(x) \\
& \wedge (\forall l: b_h \leq l \leq b_{h+1}: b_i \leq l < b_{i+1} \Rightarrow mn_i \leq (v; l \in x \dots b_{h+1}; v(x))(l) \leq mx_i) \\
& \wedge \max(mn_h, mn_{h+1}) \leq (v; l \in x \dots b_{h+1}; v(x))(b_{h+1}) \leq \min(mx_h, mx_{h+1}) \\
& \wedge b_{h+1} = b_{h+1} \\
= & \text{«predicate logic»} \\
(3.5) \quad & x \leq b_{h+1} \wedge 0 < v(x) \wedge (\forall l: b_h \leq l \leq b_{h+1}: b_i \leq l < b_{i+1} \Rightarrow mn_i \leq v(l) \leq mx_i) \\
& \wedge \max(mn_h, mn_{h+1}) \leq v(b_{h+1}) \leq \min(mx_h, mx_{h+1})
\end{aligned}$$

The first conjunct of (3.5) is implied by  $x = b_h$  in  $l$  (because, by assumption,  $b_h \leq b_{h+1}$ ). The final conjunct can be established by executing

$S_2: \text{go}(\tau); \text{await}(b_{h+1})$

where  $\tau$  is any speed that is safe and is attainable by accelerating from  $v(b_h)$ . That is,  $\tau$  must satisfy:

$$(3.6) \quad \max(mn_h, mn_{h+1}) \leq \tau \leq \min(mx_h, mx_{h+1}) \wedge \text{Vlen}(v(b_h), \tau) \leq b_{h+1} - b_h$$

Nothing stated thus far implies that it should be possible to accelerate from any safe  $v(b_h)$  to a safe  $v(b_{h+1})$  in at most a distance of  $b_{h+1} - b_h$ , and so without making further assumptions about speed constraints, the problem is unsolvable. We have uncovered another hidden assumption required to control a train:

**Assumption 2.**  $(\forall i, s: 0 < i < n \wedge \max(mn_{i-1}, mn_i) \leq s \leq \min(mx_{i-1}, mx_i);$   
 $(\exists s': \max(mn_i, mn_{i+1}) \leq s' \leq \min(mx_i, mx_{i+1}); Vlen(s, s') \leq b_{i+1} - b_i))$

Henceforth, we assume that speed constraints for blocks do satisfy Assumption 2. (It is not difficult to prove that any control problem for which there is a safe path from  $b_0$  to  $b_n$  can always be reformulated as one with more restrictive minimum and maximum speeds satisfying Assumption 2.)

A target speed  $\tau$  satisfying (3.6) can now be computed as follows. First, due to the definition of  $Vlen(u, t)$ , the set of attainable speeds  $s$ —both safe and unsafe—starting from position  $b_h$  is characterized by:

$$\sqrt{v(b_h)^2 - 2 \cdot ACC \cdot (b_{h+1} - b_h)} \leq s \leq \sqrt{v(b_h)^2 + 2 \cdot ACC \cdot (b_{h+1} - b_h)}$$

Second, the set of safe speeds  $s$  for location  $b_{h+1}$  is given by:

$$\max(mn_h, mn_{h+1}) \leq s \leq \min(mx_h, mx_{h+1})$$

The intersection of these sets, therefore, is the set of safe and attainable speeds; the maximum of this intersection is the greatest safe speed—time is money for a railroad.

$$\tau = \min(\sqrt{v(b_h)^2 + 2 \cdot ACC \cdot (b_{h+1} - b_h)}, mx_h, mx_{h+1})$$

Using this value of  $\tau$  for the target speed ensures that the final conjunct of (3.5) will hold.

The penultimate conjunct of (3.5) now is implied by our choice of  $\tau$  and the fact that  $mn_h \leq v(b_h) \leq \min(mx_h, mx_{h+1})$ , which is implied by  $Safe(b_0, b_h)$ . Thus, our only remaining obligation is the truth of the second conjunct of (3.5),  $0 < v(x)$ . Recall that  $0 \leq mn_i < mx_i$ , by assumption. Thus, for all  $i$ ,  $mx_i \neq 0$  and so successive values of  $\tau$  are each non-zero. Provided  $v_0 \neq 0$ , we can strengthen the loop invariant to include  $0 < v(x)$  as a conjunct. This results in the following program.

```

S: (x=b_0 \wedge v=(; b_0:v_0) \wedge mn_0 \leq v_0 < mx_0)
    h := 0
    (I: Safe(b_0, b_h) \wedge x=b_h \wedge 0 \leq h \leq n \wedge 0 < v(x))
    do h := n → (I \wedge h := n)
        S': r := min(sqrt(v(b_h)^2 + 2 * ACC * (b_{h+1} - b_h)), mx_h, mx_{h+1});
        go(i);
        await(b_{h+1});
        h := h + 1
        (I)
    od (h = n \wedge I)
    (Safe(b_0, b_h))

```

As the final step of the derivation, we delete references to reality variables from program statements. Recall, reality variables are auxiliary and, therefore, may not affect program execution. The only reference to a reality variable from within statements in the program above is the expression  $v(b_h)$ . We can maintain this value in a program variable  $vel$  by strengthening the loop invariant and

adding assignments after each go statement. Making these changes results in the following control program, it solves the railroad control problem.

```

S:  $\{x = b_0 \wedge v = (; b_0; v_0) \wedge mn_0 \leq v_0 < mx_0\}$ 
   $h := 0; vel := v_0$ 
   $\{I: Safe(b_0, b_h) \wedge x = b_h \wedge 0 \leq h \leq n \wedge 0 < v(x) \wedge vel = v(b_h)\}$ 
  do  $h \neq n \rightarrow \{I \wedge h \neq n\}$ 
     $S_2: t := \min(\sqrt{vel^2 + 2 \cdot ACC \cdot (b_{h+1} - b_h)}, mx_h, mx_{h+1});$ 
    go(t);
    vel := t;
    await(b_{h+1});
    h := h + 1
     $\{I\}$ 
  od  $\{h = n \wedge I\}$ 
   $\{Safe(b_0, b_h)\}$ 

```

Note that, although correct, this control program does not always permit a train to travel as quickly as possible. Modifying the derivation to maximize train speed is not difficult, however. Instead of first attempting to establish the final conjunct of (3.4), we instead concentrate on the penultimate conjunct. The loop body that results when this strategy is pursued turns out to be somewhat different from the one we derived. The alternative loop body is:

$go(t_1); await(b_{h+1} - Vlen(t_1, t_2)); go(t_2); h := h + 1$

where  $t_1$  and  $t_2$  are the largest speeds satisfying:

$$\begin{aligned} Vlen(v(b_h), t_1) + Vlen(t_1, t_2) &\leq b_{h+1} - b_h \wedge mn_h \leq t_1 \leq mx_h \\ &\wedge \max(mn_h, mn_{h+1}) \leq t_2 \leq \min(mx_h, mx_{h+1}) \end{aligned}$$

Second, note that our control programs can be easily modified for the case where the assignments to  $t$  and  $h$  are not instantaneous. The predicate logic details become a bit messier, but nothing of substance changes.

#### 4. Discussion

We were pleased to discover that only minor modifications were needed in order to employ Dijkstra's weakest-precondition calculus in deriving sequential process-control programs. Dijkstra's calculus, unfortunately, is based on regarding a program as a relation between sets of states and, therefore, does not scale-up well to concurrent and distributed programs, which are best thought of as "invariant maintainers". The extensions derived in section 2 for handling the state of a physical process do scale up. For example, we have been able to use them along with a logic for proving arbitrary safety properties of concurrent programs, Proof Outline Logic [SA86].

Reality variables are history variables—they encode in the current program state information about past system states. Using history variables for reasoning about programs is usually a bad idea, because it introduces distinctions that should be irrelevant. The current state—not how it was

computed—should be of concern when reasoning about what a program will do next. In reasoning about process-control systems, however, one has no choice but to employ history variables of some sort. This is because the past instants for which the state of a physical process is defined is a strict superset of the past instants for which the state of a control program is defined. A program implements a discrete transition system, while a physical process is likely to implement a continuous transition system. History variables allow us to reason about all of the behavior of the physical process, including those states that exist while the program state is in transition, hence undefined.

### Acknowledgments

We are grateful to Navin Budhiraja for his careful reading of an earlier draft of this paper.

### References

- [C73] Clint, M. Program proving: Coroutines. *Acta Informatica* 2, 1 (1973), 50-63.
- [G84] Gries, D. *The Science of Programming*. Springer-Verlag, New York, 1984.
- [D76] Dijkstra, E.W. *A Discipline of Programming*. Prentice Hall, N.J., 1976.
- [LP83] Lozano-Perez, T. Spatial planning: A configuration space approach. *IEEE Trans. on Computers* C-32, 2, 1983, 108-120.
- [SA86] Schneider, F.B. and G. R. Andrews. Concepts for concurrent programming. In *Current Trends in Concurrency*. (J.W. de Bakker, W.P. de Roever, and G. Rozenberg, eds.) Lecture Notes in Computer Science, Volume 224, Springer-Verlag, New York, 1986, 669-716.